Recursive Backtracking

THURSDAY, MAY 1

Section materials curated by Jonathan Coronado, Yasmine Alonso, and Sean Szumlanski, drawing upon materials from previous quarters.

This week's section exercises continue our exploration of recursion to tackle even more challenging and interesting problems. In particular, this week's section problems center around recursive backtracking, a very powerful and versatile problem solving technique.

Remember that every week we will also be releasing a Qt Creator project containing starter code and testing infrastructure for that week's section problems. When a problem name is followed by the name of a **.cpp** file, that means you can practice writing the code for that problem in the named file of the Qt Creator project. Here is the zip of the section starter code:

📦 Starter project

1. Circle of Life (`Circle.h/.cpp`)

Topics: Classes

Write a class named **`Circle`** that stores information about a circle. Your class must implement the following public interface:

```
class Circle {
        // constructs a new circle with the given radius
        Circle(double r);
        // returns the area occupied by the circle
        double area() const;
        // returns the distance around the circle
        double circumference() const;

        // returns the radius as a real number
        double getRadius() const;
        // returns a string representation such as "Circle{radius=2.5}"
        string toString() const;
};
```

You are free to add any private member variables or methods that you think are necessary. It might help you to know that there is a global constant **`PI`** storing the approximate value of π, roughly **`3.14159`**.

Solution

```
// .h file starts
#pragma once

class Circle {
public:
        Circle(double radius);

        double area() const;
        double circumference() const;
        double getRadius() const;
        string toString() const;
private:
        double r;
}

// .h file ends


// .cpp file starts
#include "Circle.h"

using namespace std;

Circle::Circle(double radius) {
        r = radius;
}

double Circle::area() const{
        return PI * r * r;
}

double Circle::circumference() const{
        return 2 * PI * r;
}

double Circle::getRadius() const{
        return r;
}

string Circle::toString() const{
        return string("Circle{radius=") + realToString(r) +
string("}");
}
// .cpp file ends
```

---

2. Compound Words (**CompoundWords.cpp**)

This question is all about splitting strings apart into smaller, nonempty pieces. To begin, we'd like you to implement a function

```
Set<Vector<string>> splitsOf(string str);
```

that takes as input a string, then returns all ways of splitting that string into a sequence of nonempty strings called **pieces**. For example, given the string "RUBY", you'd return a **`Set`** containing these **`Vector<string>`**s; notice that all letters are in the same relative order as in the original string:

```
{"R", "U", "B", "Y"}
{"R", "U", "BY"}
{"R", "UB", "Y"}
{"R", "UBY"}
{"RU", "B", "Y"}
{"RU", "BY"}
{"RUB", "Y"}
{"RUBY"}
```

If you take any one of these **`Vector`**s and glue the pieces together from left to right, you'll get back the original string **`"RUBY"`**. Moreover, every possible way of splitting **`"RUBY"`** into pieces is included here. Each character from the original string will end up in exactly one piece, and no pieces are empty.

Given the string **`"TOPAZ"`**, you'd return a **`Set`** containing all of these **`Vector<string>`**s:

```
{"T", "O", "P", "A", "Z"}
{"T", "O", "P", "AZ"}
{"T", "O", "PA", "Z"}
{"T", "O", "PAZ"}
{"T", "OP", "A", "Z"}
{"T", "OP", "AZ"}
{"T", "OPA", "Z"}
{"T", "OPAZ"}
{"TO", "P", "A", "Z"}
{"TO", "P", "AZ"}
{"TO", "PA", "Z"}
{"TO", "PAZ"}
{"TOP", "A", "Z"}
{"TOP", "AZ"}
{"TOPA", "Z"}
{"TOPAZ"}
```

As before, notice that picking one of these **`Vector`**s and gluing the pieces in that **`Vector`** back together will always give you "TOPAZ".

The decision tree for listing subsets is found by repeatedly considering answers to questions of the form "should I include or exclude this element?" The decision tree for listing permutations is found by repeatedly considering answers to questions of the form "which element should I choose next?" In this problem, the decision tree is found by repeatedly considering answers to this question:

How many characters from the front of the string should I include in the next piece?

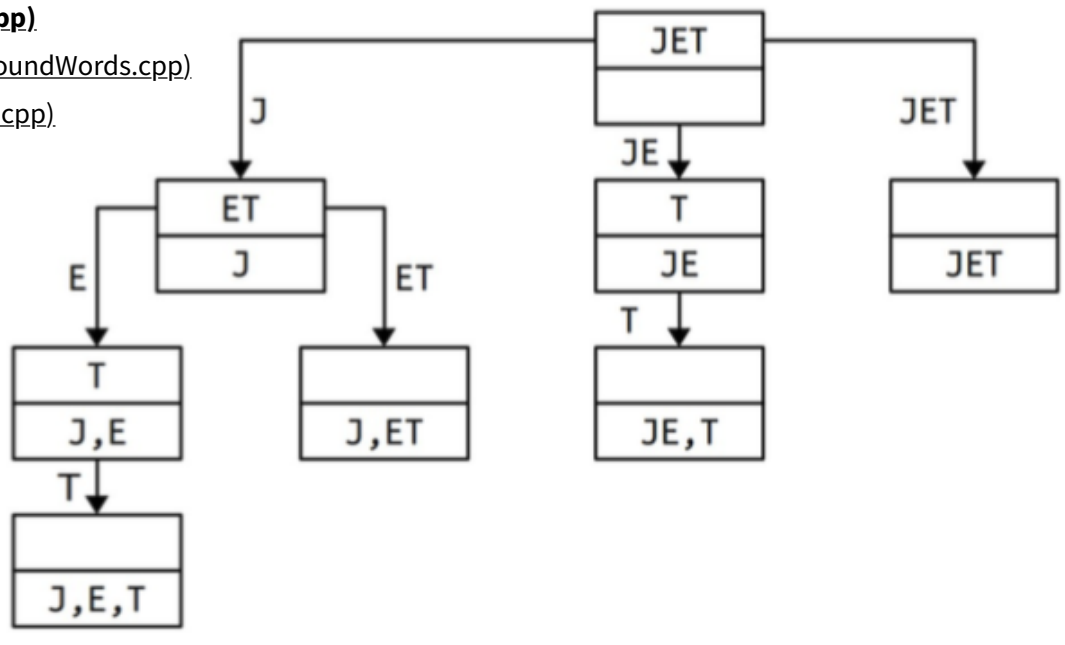
i. Decision Tree

Based on this insight, draw the decision tree for listing all splits of the string **`"JET"`** along the lines of the decision trees we drew in class. At a minimum, please be sure to do the following:

- Label each entry in the decision tree with the arguments to the recursive call it corresponds to.
- Label each arrow in the decision tree with what choice it corresponds to.

Solution

Here's the decision tree for all splits of "JET":



Here, each decision is of the form “how many characters are we taking off the front of the string?,” and we pass down through the recursion both the characters we haven’t used yet and the split we’ve built up so far.

ii. Implement `splitsOf`

Now, implement the `splitsOf` function. Your should match the decision tree that you drew in the first part of this problem.

Solution

Here’s one implementation of splitsOf based on the above decision tree:

```
Set<Vector<string>> splitsRec(string str, Vector<string>& chosen) {
    /* Base case: If we've already used up all the characters of the
input string,
     * then we've already built up one option. The set of all possible
options we
     * can make given that we're locked into this one solution is just
the set of
     * that solution by itself.
     */
    if (str == "") {
        return { chosen };
    }
    /* Otherwise, there are characters that need to get put into a
piece of the
     * split. Try all ways of doing this.
     */
    else {
        Set<Vector<string>> result;
        /* Munch off between 1 and all the characters, inclusive. */
        for (int i = 1; i <= str.length(); i++) {
            string piece = str.substr(0, i);
            string remaining = str.substr(i);

            /* Find all the splits we can make, assuming we commit to
having this
             * piece in front.
             */
            chosen.add(piece);
            result += splitsRec(remaining, chosen);
            chosen.remove(chosen.size() - 1);
        }
        return result;
    }
}

Set<Vector<string>> splitsOf(string str) {
    Vector<string> chosen = {};
    return splitsRec(str, chosen);
}
```

iii. Implement `isCompoundWord`

Now that you've gotten a warmup in, let's take aim at a related question. A ***compound word*** is an English word like "doorbell" or "heretofore" that can be split apart into multiple English words ("doorbell" becomes "door" and "bell;" "heretofore" becomes "here," "to," and "fore.") Your task is to write a function

```
bool isCompoundWord(string word, Lexicon& english);
```

that takes as input a word, then returns whether it's a compound word.

You will likely want to build on the work you've done earlier in this problem. A few things have changed, though. First, you have to split the word apart into multiple pieces. Second, each piece must itself be a word. There are many ways to account for this; explore and see what you find!

Solution

Perhaps the trickiest bit here is handling the fact that we have to split into two or more words. There are many ways to do this. Our strategy will be the following. Our recursive

logic will completely ignore this requirement, and will be perfectly content to split a string and write it as a single word if it feels like it. To force the function to always require at least one word to be munched off, we'll have our wrapper function do an initial splitting step in which it isn't permitted to munch all the characters. Here's what that looks like:

```
/* Can the given input string be split apart into zero or more English
words? */
bool isCompoundRec(string str, Lexicon& english) {
    /* Base Case: If the string is empty, we can write it as zero words
     * all glued together.
     */
    if (str == "") {
        return true;
    }
    /* Recursive Case: Split off some number of characters from the
front that
     * form a word, then recursively see if what's left is a compound
word.
     */
    else {
        /* Grab between one and all of the characters from the front.
*/
        for (int i = 1; i <= str.length(); i++) {
            /* This is a good split if what we've grabbed is a word and
the
             * remainder is an English word.
             */
            if (english.contains(str.substr(0, i)) &&
                isCompoundRec(str.substr(i), english)) {
                return true;
            }
        }

        /* Oops, nothing works. */
        return false;
    }
}

bool isCompoundWord(string word, Lexicon& english) {
    /* If we aren't a word, then we aren't a compound word. */
    if (!english.contains(word)) return false;

    /* Otherwise, try pulling off a word from the front and seeing if
what's
     * left can be made from gluing one or more words together.
     */
    for (int i = 1; i < word.length(); i++) {
        if (english.contains(word.substr(0, i)) &&
            isCompoundRec(word.substr(i), english)) {
            return true;
        }
    }

    /* Nope, it's not a compound word, since we tried every option. */
    return false;
}
```

3. Domino Tiling (**Dominoes.cpp**)

Teal: Recursive backtracking

Imagine you have a 2 × n grid that you'd like to cover using 2 × 1 dominoes. The dominoes need to be completely contained within the grid (so they can't hang over the sides), can't overlap, and have to be at 90° angles (so you can't have diagonal or tilted tiles). There's exactly one way to tile a 2 × 1 grid this way, exactly two ways to tile a 2 × 2 grid this way, and exactly three ways to tile a 2 × 3 grid this way (can you see what they are?) Write a recursive function

```
int numWaysToTile(int n)
```

that, given a number n, returns the number of ways you can tile a 2 × n grid with 2 × 1 dominoes.

Solution

If you draw out a couple of sample tilings, you might notice that every tiling either starts with a single vertical domino or with two horizontal dominoes. That means that the number of ways to tile a `2 × n` (for `n ≥ 2`) is given by the number of ways to tile a `2 × (n - 1)` grid (because any of them can be extended into a `2 × n` grid by adding a vertical domino) plus the number of ways to tile a `2 × (n - 2)` grid (because any of them can be extended into a `2 × n` grid by adding two horizontal dominoes). From there the question is how to compute this. You could do this with regular recursion, like this:

```
int numWaysToTile(int n) {
    /* There's one way to tile a 2 x 0 grid: put down no dominoes. */
    if (n == 0) return 1;

    /* There's one way to tile a 2 x 1 grid: put down a vertical domino. */
    if (n == 1) return 1;

    /* Recursive case: Use the above insight. */
    return numWaysToTile(n - 1) + numWaysToTile(n - 2);
}
```